Rover Example

A Simple Sample Application (The Planetary Rover)

This tutorial will step you through a simple EUROPA application to provide you with a solid understanding of the capabilities of EUROPA and how to approach modeling application domains. We begin by introducing the goal before stepping through the stages of creating a EUROPA application for it.

Note that this example has much in common with the nddl snippets used on the NDDL Reference page. EUROPA includes working code for this example in this directory.

Overview of the Simple Planetary Rover Application

NASA's Mars Exploratory Rover (MER) mission is operating two rovers on the surface of Mars. The rovers are moving between points of interest identified by scientists on earth to collect a range of scientific data by placing instruments and then transmitting the produced data back to earth. We are going to write the planning application for controlling a simplified version of these rovers.

To provide some context, the figure below shows NASA Ames' K9 rover that is used to experiment with advanced control concepts for future Mars rover missions. K9 is operating in a simulated Martian landscape (called Marscape) where it mirrors the MER mission. The first picture shows K9 in the context of its environments. The second picture shows K9 placing its sensor on a rock.

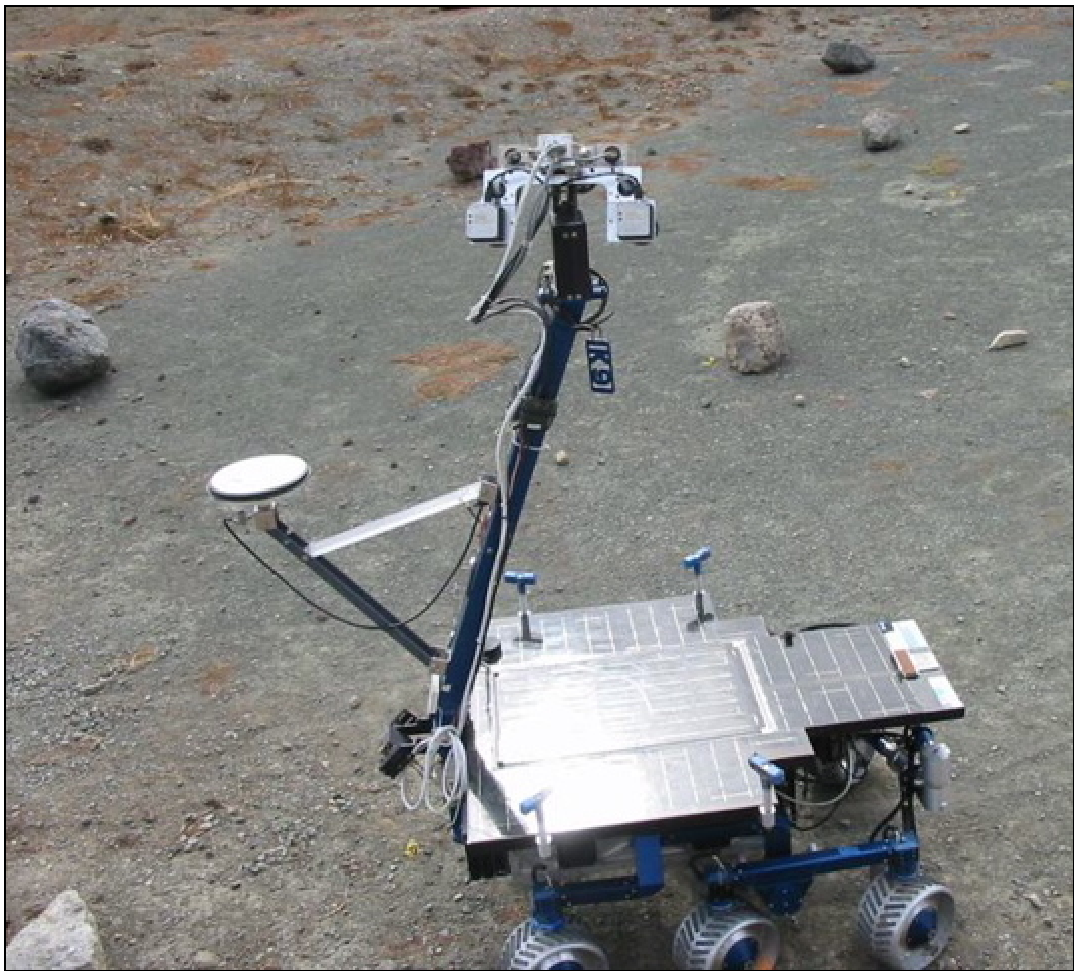


Figure 1: K9 Rover at NASA Ames' Marscape while controlled by a EUROPA planner.

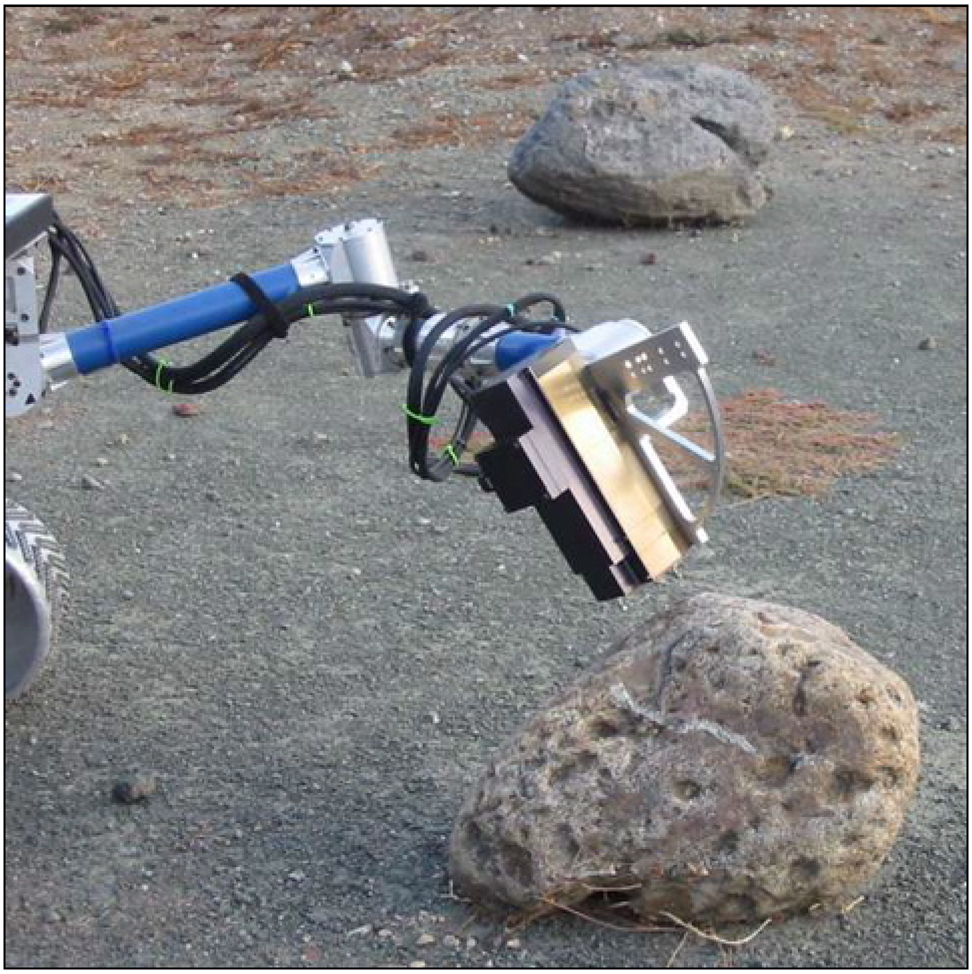
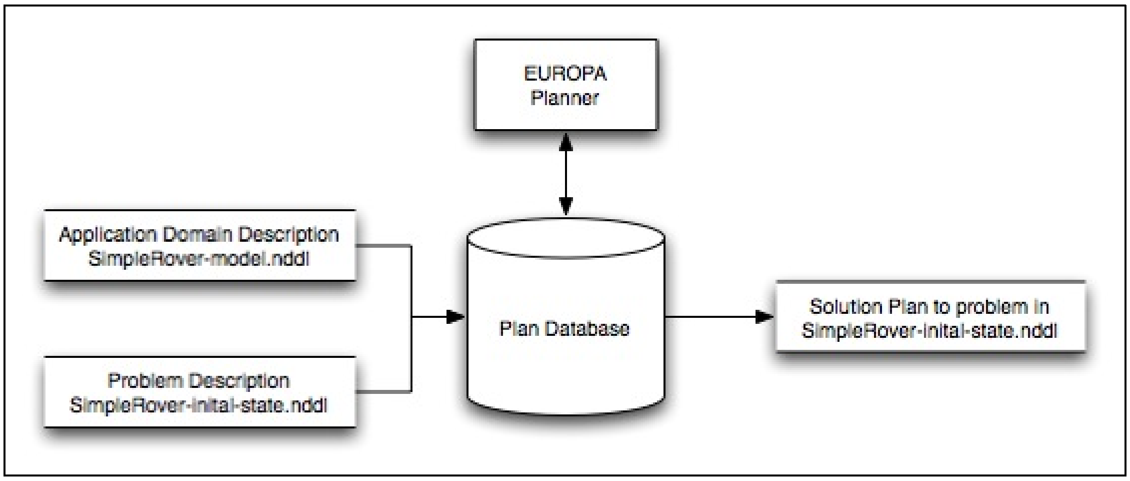


Figure 2: K9 Rover Placing a sensor at NASA Ames' Marscape.

We will build a batch application where we provide an application domain model and problem definition to the planner. The planner must then generate a plan to solve the problem. The figure below shows the main inputs and outputs to our application.


EUROPA Planner
Application Domain Description SimpleRover-model.nddl
Plan Database
Solution Plan to problem in SimpleRover-inital-state.nddl
Problem Description SimpleRover-inital-state.nddl
Figure 3: Application overview

Application Domain Analysis

Developing EUROPA applications is a design task that requires judgment and multiple iterations. We have found the steps in this tutorial useful in practice. The overall approach is to gradually build up a domain description adding detail methodically. This approach controls complexity by allowing the domain writer to focus only on well defined issues at a given instant.

The first stage is to draw a concept map of the entities in the application domain and their relationships. The figure below shows our concept map for the rover domain. We have identified locations and the paths between them as the key environment entities. The rover itself has been divided in subcomponents. The navigator concept manages the location of the rover. The instrument concept will manage the instruments for sampling rocks and the commands concept looks after instructions from the scientists that the rover will serve. Finally the battery concept is included to provide a place for managing the power used by the other components of the rover. The Lander corresponds to the vehicle used to deliver the rover to the planet surface. The Lander also provides a communication service that the rover can use to transmit information to earth.

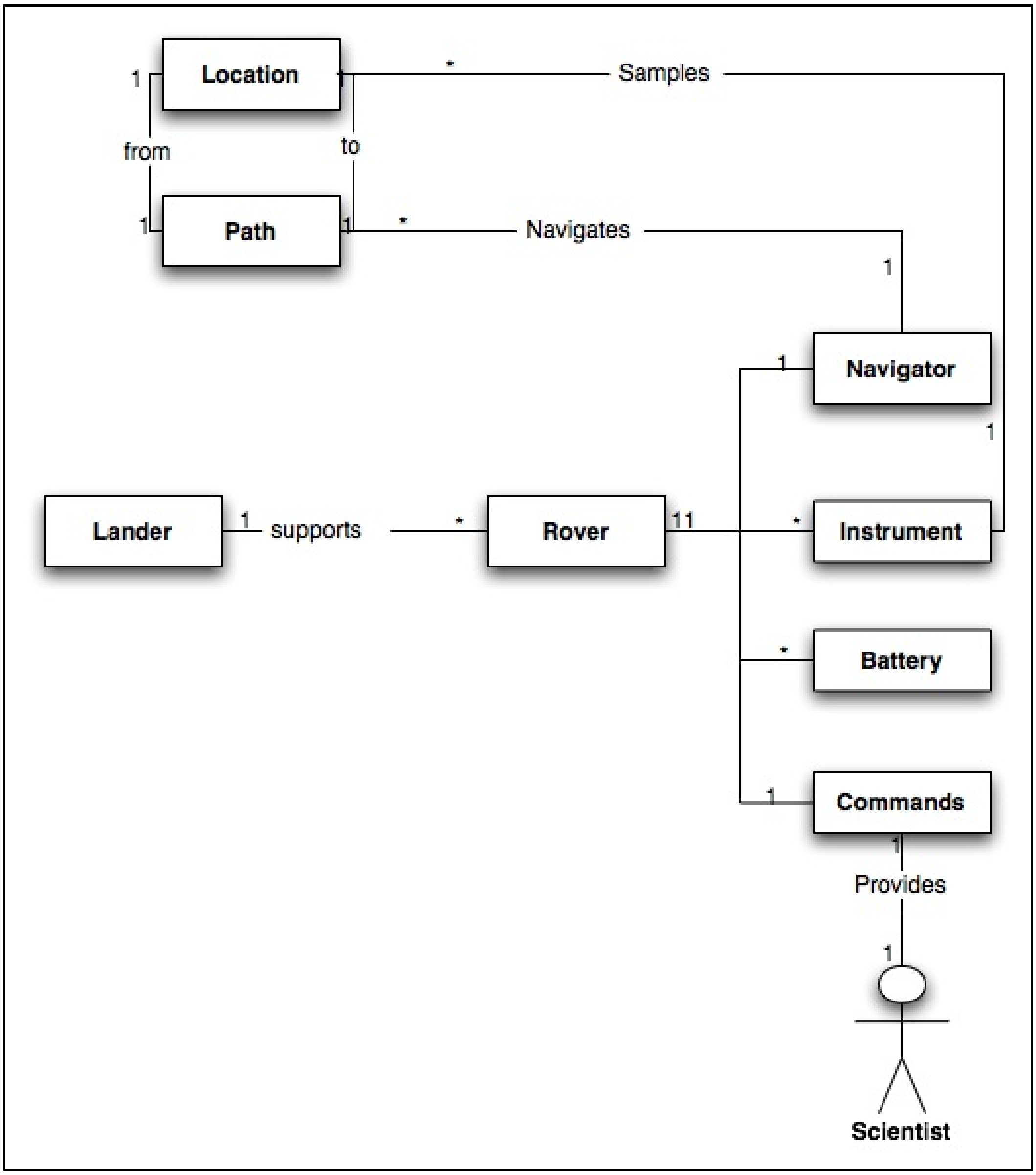


Figure 4: Rover Application Concept Diagram

The next decision is to identify the entities in our concept map that will describe changes in state of the rover as it moves around its environment and performs experiments. We call each entity a *timeline*. The concept is best introduced by example. The rover in our domain is the actor we will be planning for and will contain all the timelines. Analyzing the components of the rover produces the following breakdown of timelines:

- **Navigator** controls the rover's movement between locations and holds position at a location.
- **Instrument** controls the scientific instrument on the rover.
- **Commands** manages instructions from the scientists tasking the rover.

The next stage is to identify the states that each timeline can be in. We call each state a *predicate*. The easiest way to identify the predicates is to think through the lifecycle of each timeline. The following figure shows the set of predicates we have identified on each timeline.

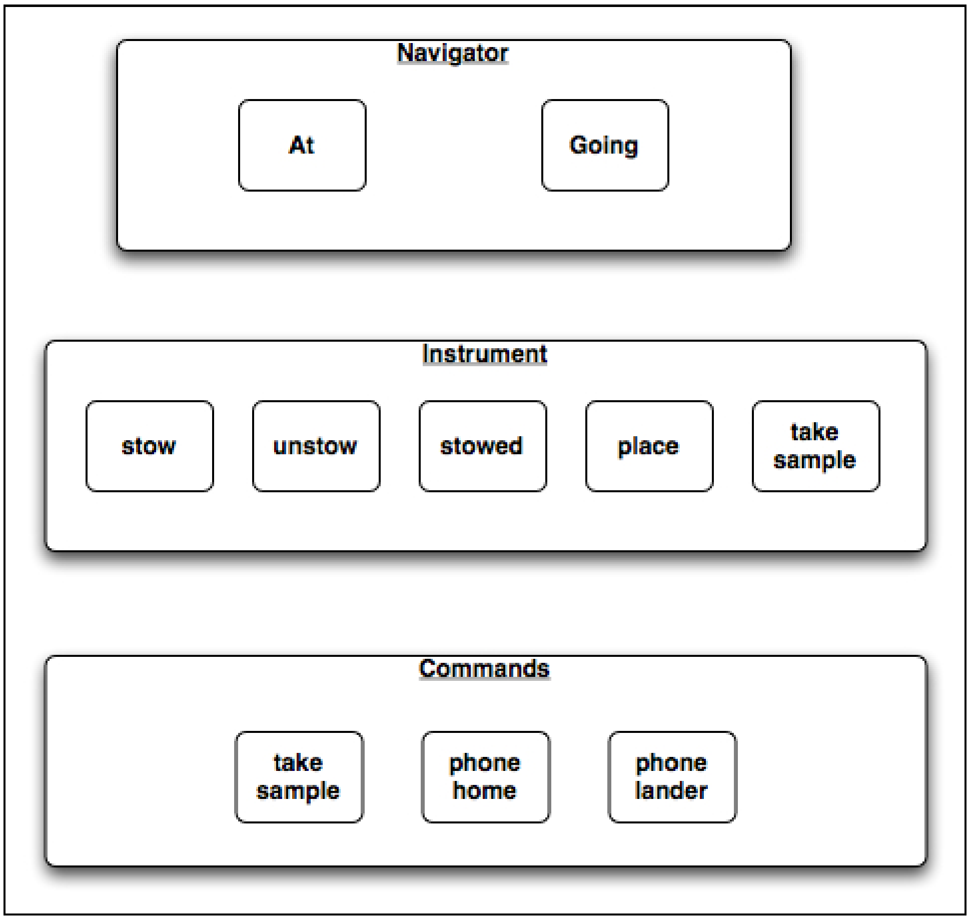


Figure 5: Initial Timelines and Predicates

Working from the top of the figure downwards:

- **Navigator:** The rover is *at* a location or it *going* between locations.
- **Instrument:** The instrument can be *stowed* away and can be in the state of being *stow*ed or being *unstow*ed. The instrument may also be *placed* on a target where it can then take a sample.
- **Commands:** The rover can be instructed to take a sample or to upload its data to the lander or directly to earth.

As noted earlier, identifying the timelines and predicates is an iterative process. It would not be unusual to find new timelines while identifying predicates or to discover that two timelines could be collapsed into one. Please iterate and be comfortable with experimentation.

The next stage is to flesh out the properties of the predicates and the constraints between them. Lets begin by adding these elements to the figure above (we've omitted the properties of the instrument's predicates for clarity). We focus first on just the transitions between predicates within a timeline not the constraints between predicates on different timelines. The black circle indicates a startup state for the timeline while the black circle with an outer circle indicates a terminal state.

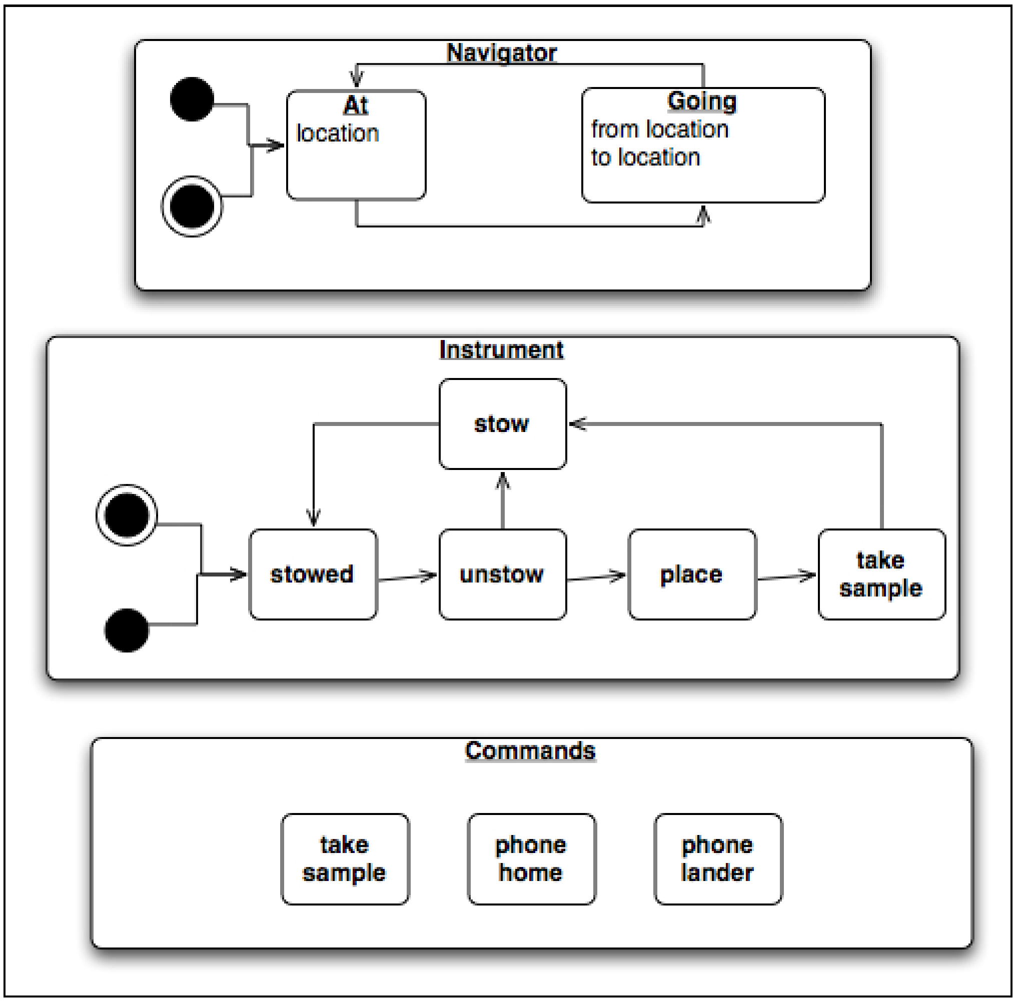


Figure 6: Timelines and Predicates with Transitions between Predicates on each Timeline.

The meaning of the state transitions added to this figure are intuitive. A timeline is always in a given state. It can transition only to a new state connected in our diagram by an arrow. The next stage is to consider the constraints between predicates on different timelines. So far we have only used the notion of state transitions to connect predicates; these map to the temporal relations of *meets* and *met_by* and are sufficient for a timelines where only one predicate instance can occur at any given moment. When we start to connect predicates between timelines we need to use the full range of temporal relations as we begin to deal with concurrent states.

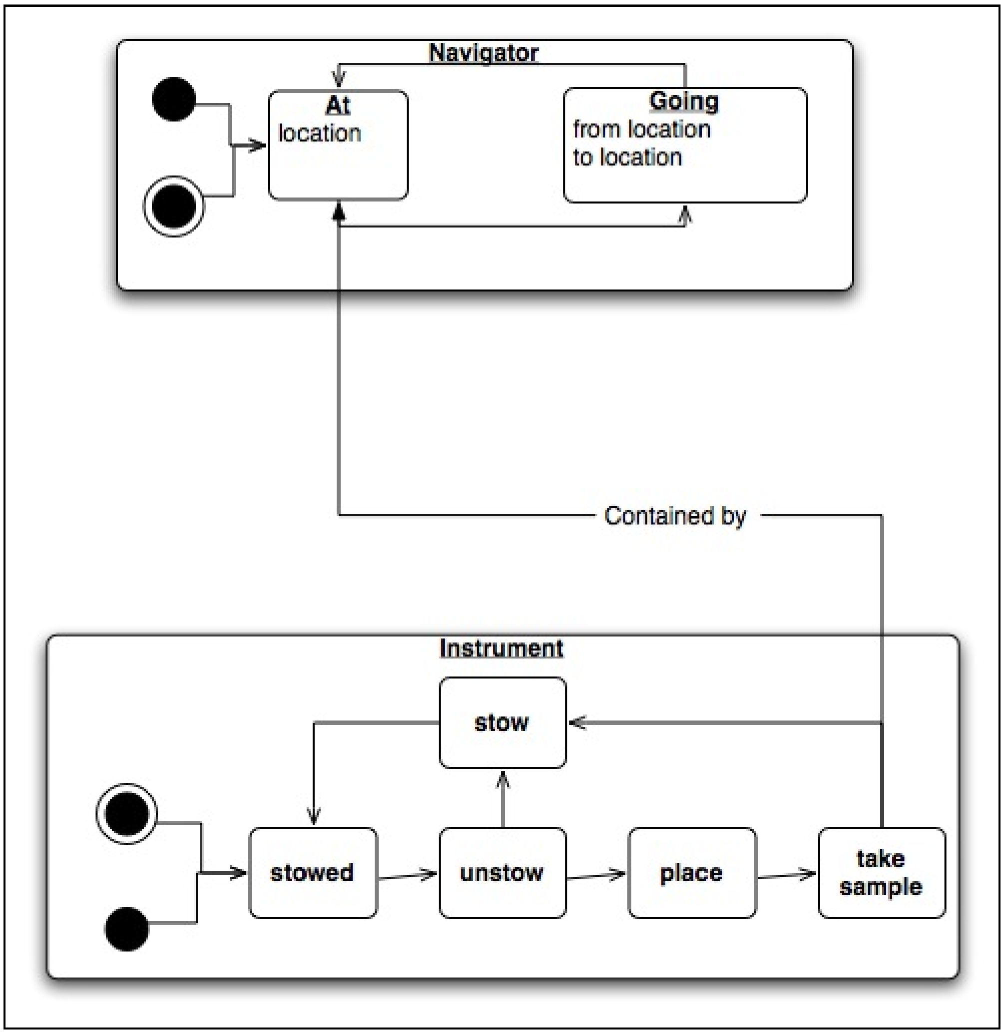


Figure 7: Iterations Between Timelines.

We have only one interaction between timelines in this application. The *take sample* predicate requires that the navigation timeline be *at* the location throughout its lifetime. We are identifying the common sense constraint that we have to be at and remain at a location in order to sample it.

We now have the main concepts in the application identified and categorized and we are ready to start encoding it in NDDL.

NDDL Encoding - Model

Our encoding of the Rover domain is available here. We will step through each of the model files and explain how it was derived form our analysis above and the NDDL constructs that were used. Rover-model.nddl contains the application domain model and Rover-inital-state.nddl contains the sample problem definitions.

We begin with the application domain model. The first two lines are include statements. NDDL allows us to develop module domain descriptions where elements are put into separate files. This allows us to reuse model elements between applications and to control the complexity by showing only related elements in a given file. In this example, we include two files that make available some default NDDL structures. For now, ignore the detail of the included files and just note that it is possible to include files.

```
 #include "Plasma.nddl"
 #include "PlannerConfig.nddl"
 #include "Resources.nddl"
```

Defining Locations

The concept of locations is encoded in the *Location* class. The class has three attributes. The *name* is a symbolic name for the location. The *x* and *y* attributes are coordinates. We have decided to position all our locations on a Euclidean Plane. The second part of the class specifies the constructor. The constructor defines how the attributes of a location are initialized when a new instance is created. We simply provide a signature that allows us to create locations with the name and coordinates specified; the constructor's job is to copy those initial values into the correct member variables.

```
class Location {
   string name;
   int x;
   int y;

   Location(string _name, int _x, int _y){
     name = _name;
     x = _x;
     y = _y;
   }
 }
```

Defining Paths

The *Path* class connects pairs of locations and its implementation follows a similar pattern to the *Location* class. Paths have a symbolic name and a pair of locations that they connect. The *cost* parameter will be used to compute the amount of battery power that will be consumed while traversing a path. The constructor's role is again just to take initial values for each of a path's attributes.

```
class Path {
  string name;
  Location from;
  Location to;
  float cost;

  Path(string _name, Location _from, Location _to, float _cost){
    name = _name;
    from = _from;
    to = _to;
    cost = _cost;
   }
 }
```

Defining Resources

We next encode the *Battery* that will be used to power the rover. EUROPA provides a *Reservoir* class for modeling consumable resources like batteries and fuel cells that have a numeric capacity that is produced and consumed during a plan. The *Battery* class specializes the general *Reservoir* class. It takes three arguments: an initial capacity *ic*, a minimum charge level *ll_min* and a maximum charge level *ll_max*. The *Battery* constructor delegates the creation of an instance to its parent or super class, the general *Reservoir* class.

```
class Battery extends Reservoir {
  Battery(float ic, float ll_min, float ll_max){
    super(ic, ll_min, ll_max);
  }
}
```

Defining the Navigator Timeline

We now move on to encode the components of the rover. We begin with the rover's navigator. This class contains the two predicates we identified earlier. The *At* predicate models the concept of the rover being at a particular location. The *Going* predicate models the concept of moving between locations. The *neq* construct is a constraint that ensures the rover does not attempt to traverse a path that start and finishes in the same location.

```
class Navigator extends Timeline {
  {
    predicate At{
    Location location;
  }

  predicate Going{
    Path p;
    Location from;
    Location to;
    neq(from, to); // prevents rover from going from a location straight back to that location.
}
```

The next element encodes the detail of the *At* predicate. The first constraint is a *met_by* that specifies instances of this predicate will be proceeded by an instance of the *Going* predicate. We name that predecessor 'from'. We then post the constraint that the 'from' predicate's *to* attribute is equal to the *location* attribute of this predicate. We are just saying that a preceding *Going* predicate must end at the location of this *At*. The third constraint specifies that an *At* predicate will be followed by a *Going* predicate. The *from* location of that succeeding Going *predicate* must be set this *location.*

```
Navigator::At{
   met_by(object.Going from);
   eq(from.to, location); // next Going token starts at this location
   meets(object.Going to);
   eq(to.from, location); // prevous Going token ends at this location
}
```

We next specify the details of the *Going* predicate. It first mirrors the *At* predicates constraints in ensuring that a *Going* is always preceded and followed by an *At* predicate and that the *to* and *from* attributes are synchronized correctly. The *Path* element models the constraint that a *Going* predicate must pass along a path that connects to locations being traversed. Finally, the battery components specifies that the change in battery level caused by a *Going* predicate is equal to the cost associated with the path selected.

```
Navigator::Going{
  met_by(object.At _from);
  eq(_from.location, from);
  meets(object.At _to);
  eq(_to.location, to);

  eq(p.from, from);
  eq(p.to, to);

  starts(Battery.consume tx);
  eq(tx.quantity, p.cost);
 }
```

Defining the Commands Timeline

The *Commands* timeline manages instructions from the scientist user to take samples (*TakeSample*) or transmit information back to the lander (*PhoneLander*) or earth (*PhoneEarth*).

```
class Commands extends Timeline {
   predicate TakeSample{
     Location rock;
     eq(duration, [20, 25]); // Flexible durations for taking a sample
   }

   predicate PhoneHome{}
   predicate PhoneLander{}
}
```

The next element details the constraint on each of the *Commands* class' predicates. Consider *TakeSample* first. Much of the definition will be familiar to you from the previous predicates. The contains constraint ensures that an *Instrument* class *TakeSample* action must occur sometime during this predicate. The next line names the rock that that sample will occur at as 'rock'. The condition allows the predicate to either meet a *PhoneHome* or *PhoneLander* predicate to transmit the results of the sampling back to earth. Either way can be used by the planner unless we set specify the value of the *OR* variable.

```
Commands::TakeSample{
   contains(Instrument.TakeSample a);
   eq(a.rock, rock);
   Rover rovers;

   bool OR;

   if(OR == false){
     meets(object.PhoneHome t0);
   }
   if(OR == true){
     meets(object.PhoneLander t1);
   }
}
```

The *PhoneHome* and *PhoneLander* predicates are specified next. Both use battery power. *PhoneHome* is more expensive as the signal needs to be transmitted back to earth requiring much more power.

```
Commands::PhoneHome{
   starts(Battery.consume tx);
   eq(tx.quantity, 600); // consume battery power
}

Commands::PhoneLander{
  starts(Battery.consume tx);
  eq(tx.quantity, 20); // consume battery power
}
```

Defining the Instrument Timeline

This timeline details the management of the rover's instruments for taking samples and for keeping the instrument safely stowed while moving. The first predicate is *TakeSample* and is constrained to take exactly ten time units.

```
class Instrument extends Timeline {
   predicate TakeSample{
   Location rock;
   eq(10, duration); // duration of TakeSample is 10 time units
}
```

Place models the process of putting the instrument on the rock once the rover has driven up to it. This is constrained to take exactly three time units.

```
predicate Place{
   Location rock;

eq(3, duration); // duration of Place is 3 time units}
```

We then model the actions for stowing the instrument safely and for the actions of stowing it and unstowing it. Both actions take exactly two time units.

```
predicate Stow{
  eq(2, duration); // duration of Stow is 2 time units
}

predicate Unstow{
   eq(2, duration); // duration of Unstow is 2 time units
}

predicate Stowed{}
```

With the predicates defined we now move on to specify the detailed constraints on each beginning with *TakeSample*. We first constrain the predicate to occur while the rover's navigator it at the location of the rock. The action must be proceeded by a Place predicate and succeeded by a Stow predicate. The predicate consumes 120 units of battery power.

```
Instrument::TakeSample{
  contained_by(Navigator.At at);
  eq(at.location, rock);
  Rover rovers;

  met_by(Place b);
  eq(b.rock, rock);

  meets(Stow c);

  starts(Battery.consume tx);
  eq(tx.quantity, 120); // consume battery power
}
```

The *Place* predicate is similar to the *TakeSample* predicate.

```
Instrument::Place{
  contained_by(Navigator.At at);
  eq(at.location, rock);
  Rover rovers;

  meets(TakeSample a);
  eq(a.rock, rock);
  met_by(Unstow b);

  starts(Battery.consume tx);
  eq(tx.quantity, 20); // consume battery power
}
```

Unstow must occur while the navigator is stationary and is proceeded by a *Stowed* predicate as it is not possible to unstow the instrument from any other state.

```
Instrument::Unstow{
  contained_by(Navigator.At at);
  Rover rovers;

  meets(Place a);
  met_by(Stowed b);

  starts(Battery.consume tx);
  eq(tx.quantity, 20); // consume battery power
}
```

The *Stow* predicate is followed by a *Stowed* predicate and succeeds a *TakeSample* predicate.

```
Instrument::Stow{
  contained_by(Navigator.At at);
  Rover rovers;

  meets(Stowed a);
  met_by(TakeSample b);

  starts(Battery.consume tx);
  eq(tx.quantity, 20);  // consume batter power
}
```

Stowed just ensures that it sits between *Stow* and *Unstow* predicates.

```
Instrument::Stowed{
  met_by(Stow a);
  meets(Unstow b);
}
```

Defining the Rover

The *Rover* class pulls together all the components we have defined so far. It has an attribute for the navigator, instrument, and battery classes. The constructor takes an instance of the *Battery* class and creates instances of the other classes to setup the rover.

```
class Rover {
  Commands commands;
  Navigator navigator;
  Instrument instrument;
  Battery mainBattery;

  Rover(Battery r){
    commands = new Commands();
    navigator = new Navigator();
    instrument = new Instrument();
    mainBattery = r;
  }
}
```

We have now completed the NDDL modeling we need to describe our application domain and it is now possible to describe problems that we want our planner to solve.

NDDL Encoding - Initial State

The initial state file contains an example planning problem with a specific set of locations and paths. The first line of the file just includes the domain model file we detailed in the previous section.

```
#include "./SimpleRover-model.nddl"
```

The *PlannerConfig* sets the planning horizon to between 0 and 100 units and gives the planner 600 search steps to search for a solution.

```
PlannerConfig world = new PlannerConfig(0, 100, 600);
```

The environment is configured to have five locations - four rocks and a single lander.

```
Location lander = new Location("LANDER", 0, 0);
Location rock1 = new Location("ROCK1", 9, 9);
Location rock2 = new Location("ROCK2", 1, 6);
Location rock3 = new Location("ROCK3", 4, 8);
Location rock4 = new Location("ROCK4", 3, 9);
```

We define three paths that lead from the lander to the location named *rock4*. The paths vary considerably in the amount of battery energy needed to traverse them.

```
Path p1 = new Path("Short Cut", lander, rock4, 400.0);
Path p2 = new Path("Very Long Way", lander, rock4, 2000.0);
Path p3 = new Path("Moderately Long Way", lander, rock4, 1500.0);
```

We define a single battery with an initial and maximum capacity of 1000 units. The battery may be drained as low as 0 units.

```
Battery battery = new Battery(1000.0, 0.0, 1000.0);
```

We define a single rover, *spirit*, and pass it the battery we just defined.

```
Rover spirit = new Rover(battery);
```

We have now defined all the objects in our domain and we can close the database and begin specifying the state of the world.

```
close();
```

We define the initial state by creating an *At* token called *initialPosition*. It is constrained to start at the same time the planning horizon starts and the location attribute is set to being the *lander*. The result is to place the rover at the lander at time zero.

```
goal(Navigator.At initialPosition);
eq(initialPosition.start, world.m_horizonStart);
eq(initialPosition.location, lander);
```

We define the goal as taking a sample of *rock4* starting at time 50.

```
goal(Commands.TakeSample sample);
sample.start.specify(50);
sample.rock.specify(rock4);
```

The initial state is completed by setting the initial state of the instrument to stowed.

```
rejectable(Instrument.Stowed stowed);
eq(stowed.start, world.m_horizonStart);
```

With the model and the initial state now specified it is time to start planning.

Results

Run the example:

```
% cd $EUROPA_HOME/examples/Rover
% ant
```

Click on "Go" in the solver dialog, then run "setupDesktop()" from the BeanShell console. You will see the resulting schedule for this particular example (windows have been rearranged in this screenshot to show everything):

Error: Macro Image(DocImgs:Rover.results2.jpg) failed

```
Attachment 'wiki:DocImgs: Rover.results2.jpg' does not exist.
```

A couple of observations:

- The blue curve in the Resources window show the battery charge over time for the selected plan. Although the battery starts with 1000 units, the curve starts at 600 since the plan requires the rover to immediately take the path that costs 400 units.
- The bottom window displays a gantt chart for the Command, Navigator and Instrument instances in this problem. Hover the mouse over any piece (green rectangle) of the gantt chart to see details displayed in the Details window. In this screenshot, the mouse was hovered over the largest green box in the Commands line. Looking at the Details pane, we can see this is the scientist's requested TakeSample command, with duration 20, start 50, and end 70.